Master's Thesis

Improving the I/O Performance of Disk-Based Graph Engine by Graph Ordering

Keunhak Lim

Department of Computer Science and Engineering

Graduate School of UNIST

2018

Improving the I/O Performance of Disk-Based Graph Engine by Graph Ordering

Keunhak Lim

Department of Computer Science and Engineering

Graduate School of UNIST

Improving the I/O Performance of Disk-Based Graph Engine by Graph Ordering

A thesis/dissertation

submitted to the Graduate School of UNIST

in partial fulfillment of the

requirements for the degree of

Master of Science

Keunhak Lim

12/14/2012

Approved by

Advisor

Sam H. Noh

Improving the I/O Performance of Disk-Based Graph Engine by Graph Ordering

Keunhak Lim

This certifies that the thesis of Keunhak Lim is approved.

12/14/2012

signature

____

Advisor: Sam H. Noh

signature

____

Beomseok Nam: Thesis Committee Member #1

signature

____

Young-ri Choi: Thesis Committee Member #2

Abstract

With the advent of big data and social networks, large-scale graph processing becomes popular research topic. Recently, an optimization technique called Gorder has been proposed to improve the performance of in-memory graph processing. This technique improves performance by optimizing the graph layout on memory to have better cache locality. However, because the Gorder was designed with only the algorithms which have a specific I/O pattern, it is not suitable for some other graph algorithms; also, the cost for applying the technique is significantly high. To solve the problem, we propose a new graph ordering called Neighborhood Ordering (N.Order). N.Order considers the characteristics of I/O accesses for SSDs and HDDs to improve the performance of disk-based graph engine. In addition, the algorithmic complexity of N.Order is simple compared to Gorder, hence it is cheaper to apply N.Order. N.Order reduces the cost of pre-processing up to 14 times compared to the Gorder, still its performance is 2 times faster compared to the random ordering.

Contents

List of Figures

List of Tables

1. Introduction

With the advent of big data and social networks, large-scale graph processing becomes popular research topic. Accordingly, various graph processing systems have been developed to process large graphs. The graph processing system is divided into a distributed cluster-based graph engine such as Pregel, PowerGraph, and GraphLab, and a disk-based single machine graph engine such as GraphChi, TurboGraph, and FlashGraph [1, 2, 3, 4, 5, 6]. Due to the cost of management of a cluster, network communication overhead, and so on, a disk-based single machine graph engine becomes a research trend these days. Therefore, in this paper, we focus on the performance improvement of disk-based graph engine which is a recent research trend.

Where should we focus on improving the performance of disk-based graph engine? According to the results of previous studies [1, 6, 7], Most of the total execution time is occupied by I/O processing time. Figure 1 shows the portion of the I/O processing time of the total execution time when the graph algorithm is executed in FlashGraph, the latest graph engine. According to these results, the I/O processing time in the disk-based graph engine occupies at least 55% of the total execution time in SSD and at least 70% of the total execution time in HDD. So, most of the total execution time is occupied by I/O processing. As a result, I/O processing performance is the main determinant of disk-based graph processing performance.

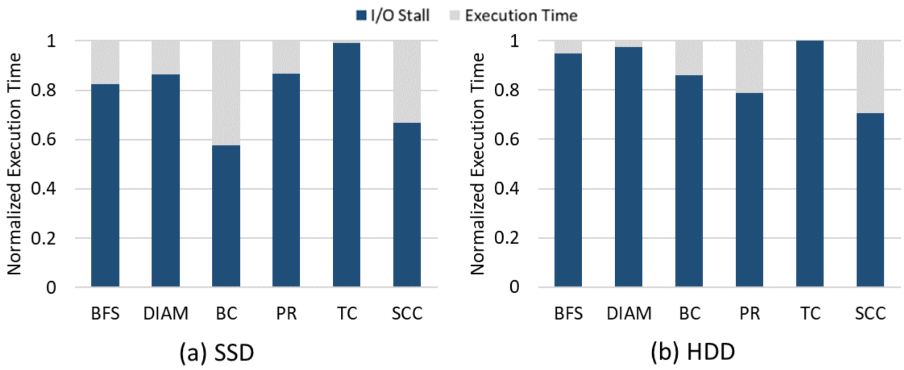


Figure 1 Ratio of I/O processing time to total execution time when executing graph algorithms in disk-based graph engine. (The page cache size is 10% of the graph size, and the algorithm is executed with a single thread)

Many techniques have been proposed to improve I/O performance in graph processing. Among them, the graph ordering improves the performance by utilizing the natural characteristics of the graph itself.

Existing graph processing studies have focused on improving systemic performance by proposing new data structures for efficient I/O management or by developing new algorithms. On the other hand, graph ordering is a general approach to improve the processing performance or graph algorithms with their implementation and data structures unchanged.

Gorder, the state of the art research, has been proposed as a prior study of performance improvement through graph ordering [8]. Gorder was designed to maximize the CPU cache locality in order to reduce CPU cache stall overhead caused by graph processing. However, since the Gorder is designed only for algorithms with limited workload patterns, the performance improvement is not significant for some algorithms. In addition, the locality scoring equation, which is calculated when the Gorder is executed, has a large overhead, so Gorder takes lots of pre-processing costs.

In this paper, we propose a new graph ordering, N.Order, with the following objectives. First, when an existing good graph algorithm is proposed, it should improve performance without changing the algorithm. Second, it should improve performance without changing the existing well-designed graph processing system implementation. Third, the overhead should be reduced by minimizing the pre-processing time. Fourth, even if the pre-processing cost is taken, the performance against the pre-processing cost should be high enough to perform the ordering. N.Order proposed in this paper can be applied to existing systems and algorithms without any changes. The pre-processing performance is improved by at least 6.2 times and up to 14 times compared with Gorder, which is the state of the art ordering technique, and at the same time, the graph processing performance is improved by at least 2 times and 10 times more than random ordering.

Major contributions: First, we analyzed the impact of graph ordering on performance. In disk-based graph processing systems, page cache hit ratio, I/O size, etc. are changed according to graph ordering, which leads to performance. Second, we analyzed the I/O patterns of the graph algorithm to classify the graph algorithms according to their characteristics. Since the I/O request patterns are different for each graph algorithm, this pattern is classified into 4 types. Third, we model the relationship between graph ordering and algorithm performance and verify its modeling. Fourth, we propose N.Order, which is less expensive than the existing graph ordering, Gorder, and performs better in some graph algorithms. The proposed scheme is designed with a very simple scheme so that the pre-processing performance is up to 14 times higher than the Gorder and at the same time the performance is up to 10 times higher than the random ordering.

Paper Organization: The composition of the paper is as follows. In Chapter 2, we introduce the outline of graph ordering and its effect on performance. In Chapter 3, we analyze the behavior of the graph algorithm in the disk-based graph engine and classify it according to the characteristics. And in Chapter 4, we model and verify graph ordering and performance to determine which graph ordering is better. In Section 5, we propose a new graph ordering method, called N.Order. The conclusions are drawn in Chapter 7. In Chapter 8, we briefly introduce the related research, the graph engine. Various experiments performed in all chapters were based on FlashGraph, a latest disk-based graph engine.

2. Graph Ordering

Graph ordering changes the order in which they are stored in storage by assigning a new ID to the vertex. Generally, in disk-based graph engine, graph data is sorted and stored in order of vertex ID, so graph layout can be changed on the storage device through graph ordering.

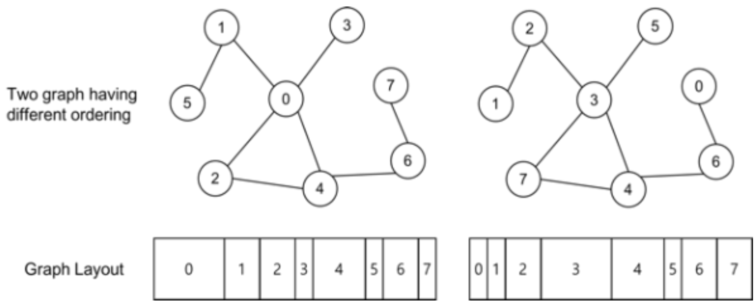


Figure 2 Graph layout with two different graph ordering.

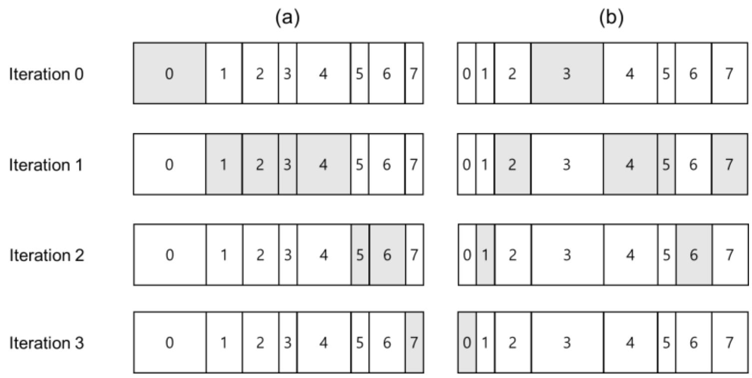


Figure 3 Disk access pattern in the BFS algorithm.

Figure 2 shows the on-disk graph layout for two different graph ordering. If the graph layout is changed through graph ordering, the I/O request pattern that occurs when executing the graph algorithm also changes. The change of I/O request pattern on the storage device affects the performance of the whole algorithm execution. Figure 3 shows the difference in disk access patterns when performing BFS algorithm on a graph with two different graph orders. In Figure 3 (a), disk access is relatively contiguous as compared to (b), so I/O patterns are relatively contiguous. The right disk access pattern generates random I/O to the disk, and this random I/O adversely affects the disk performance. In addition, if the IDs are adjacent to each other, the probability of being stored in the same page increases, and the total I/O request size is reduced.

Efficient I/O requests are very important because most of the graph processing time in the disk-based graph engine takes up I/O processing time. Graph ordering can improve graph engine performance by generating as close to the disk I/O requests as possible. However, finding optimal graph ordering is very difficult. The number of graph orderings that can be represented in a graph is |V|!. It is close to infinity. In addition, graph ordering must be completed within a short period of time because it is a cost to pay before the graph processing operation. The last thing to consider is the various graph algorithms. It is very difficult to find graph ordering suitable for all algorithms because the I/O request patterns and graph characteristics are different for each graph algorithm. Good graph ordering must meet the following three criteria.

- *Fast pre-processing (ordering)*
- *Enough performance improvements*
- *Portability applicable to various graph algorithms*

To find a better graph ordering, we need to understand the disk access pattern of the graph algorithm and find out the layout of the graph that suits processing performance. The next section discusses the behavior and characteristics of graph algorithms on disk-based graph engines.

3. Characteristics of Graph Algorithms

3.1. Process of Graph Computation

In the vertex-centric programming model, a graph algorithm is executed by calling a vertex function defined in each vertex. When the vertex function is called, the graph engine reads the data and the edge list from the disk for the operation of the vertex. While the vertex function is executing, the neighbor vertex is activated through the edge list of the vertex, and the vertex is deactivated. The vertex function of the activated neighbor vertex is called again, and it is repeated until all vertices are deactivated [1, 9].

Since calls to vertex functions propagate to neighbor vertices through the edges, disk I/O in the disk-based graph engine also propagates through the edges. This computation procedure can be represented by the following general statement. This statement is also commonly used to the Gorder [8].

for each node $v \in N_O(u)$ **do**
    the program segment to compute/access v

These arithmetic patterns are common patterns in various graph algorithms. In the next section, we will analyze the I/O patterns of graph algorithms and propose a performance modeling that can estimate various existing graph orderings. Then, we propose a new graph ordering that can improve the performance of disk-based graph engine through this modeling.

Limitations: In the following section, we do not consider the case of the graph algorithm that is different from the computation procedure assumed in this chapter. For example, in the case of PageRank, the order of operations may be invoked in the vertex ID order rather than propagating through the edge. This case is not considered because it violates this computation procedure.

3.2 Graph Algorithms on Disk-Based Graph Engines

When performing graph algorithms in the vertex-centric programming model, there are differences in disk access workloads depending on the algorithm. In the previous chapter, we assumed that I/O requests are propagated along the edges during graph algorithm computation. Under these assumptions, we can generally deduce that the higher the in-degree, the more I/O requests will be made. If this is true, would not it be better to assign a vertex ID in in-degree order? Figure 4 shows the in-degree and the number of I/O requests for each vertex when performing Breadth-First Search (BFS), Diameter (DIAM), Betweenness Centrality (BC), PageRank (PR) and Triangle Counting (TC) algorithms. According to this figure, the number of I/O requests is constant regardless of the degree in BFS, DIAM, BC, and PR. Conversely, in the case of TC, the in-degree of the vertex and the number of I/O requests are relatively proportional, but the number of I/O requests is very small compared to the in-degree. Therefore, it is not a good idea to allocate new ID in order of in-degree.

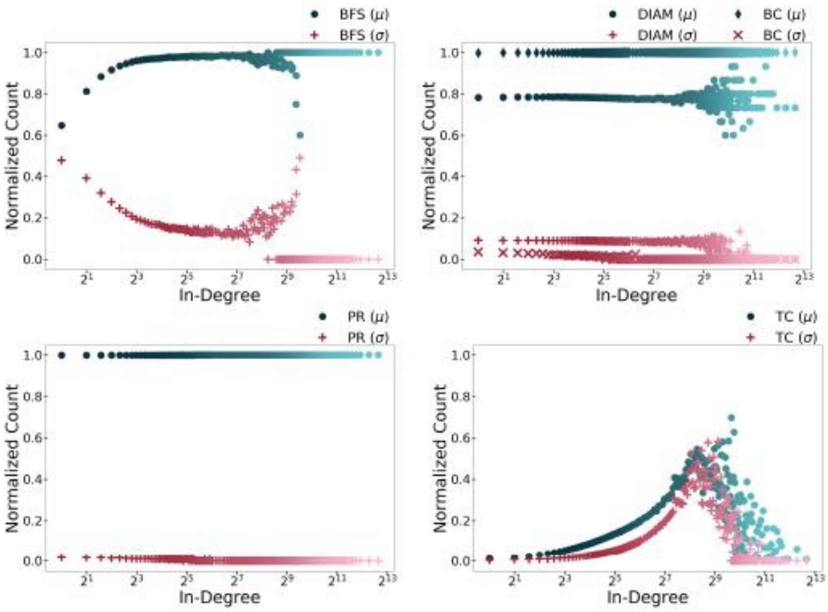


Figure 4 Relationship between the in-degree of vertex and the number of I / O requests during graph algorithm execution.

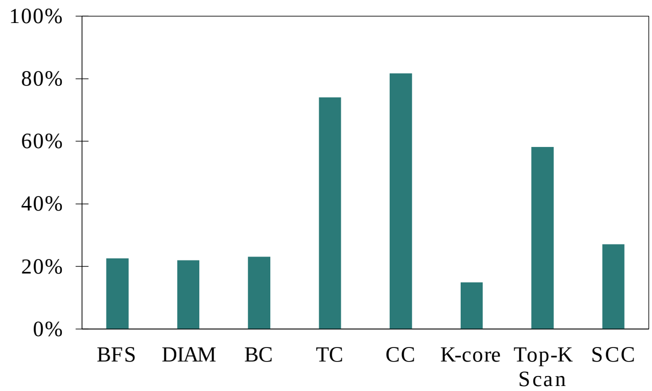


Figure 5 Page cache hit ratio of various graph algorithms.

The difference in the characteristics of the algorithm shown in Figure 4 leads to the difference in page cache hit ratio. Figure 5 shows the page cache hit ratio of each graph algorithm. When the BFS is executed, the vertices which generate the disk request already once do not generate the I/O request again. Therefore, the page cache hit ratio is measured very low. Conversely, algorithms such as TC have a relatively high cache hit ratio. This is because a certain vertex and neighboring vertices generates a disk I/O requests repeatedly when executing TC. That is, since the same data is repeatedly accessed, the cache hit ratio is high.

It is very difficult to find an optimal graph ordering that reflects these various features because it has a very diverse workload for each graph algorithm. When assigning a vertex ID from Gorder, the latest graph ordering technique, it is assigned using the following scoring equation (1).

$$S(u,v) = S_s(u,v) + S_n(u,v) \tag{1}$$

However, the locality of all algorithms is not expressed in that way. Figure 6 shows the intermediate process of graph ordering. After the current 5 vertices have been assigned, the 6th vertex must be selected according to the scoring result in Gorder. Is this the right choice? It is not always the case. This ordering is a limited graph ordering only suitable for a highly localized graph algorithm that requests I/O several times in units of one vertex such as TC and CC.

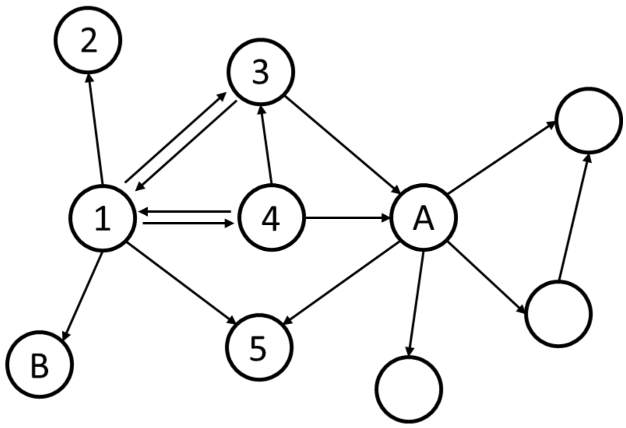


Figure 6 Which vertex is better?

Some algorithms, such as BFS, DIAM, and BC, are not suitable for this. These algorithms perform neighborhood vertices rather than the vertices calculated by locality score. In Figure 6, when processing the BFS algorithm, it is better to assign the vertex B for the 6th vertex, which is the neighbor vertex directly connected to the vertex 1 than the vertex A, preferentially. In other words, it is more appropriate to allocate in a manner that is propagated to neighboring vertices rather than scoring scheme allocation.

Therefore, in this paper, we propose Neighborhood Ordering (N.Order), which is a novel graph ordering that allocates neighbors vertices first. In low-locality graph algorithms such as BFS, DIAM, and BC, it is most important to locate adjacent vertices that require I/O requests. N.Order is designed to improve the performance of this type of algorithms by focusing on allocating adjacent vertices. Details of N.Order are discussed in Chapter 6.

3.3 Classification of Graph Algorithms

According to the results of the analysis in the previous section, we classify graph algorithms according to their characteristics. Figure 7 shows the classification of the graph algorithm according to the disk access pattern. The diagram on the left is the Neighborhood-First algorithm and the right is the Locality-First algorithm. The algorithm outside the diagram is an algorithm in which the execution order of vertices is not along the edges, that is, it is invoked in the order of the vertices IDs, regardless of the graph topology.

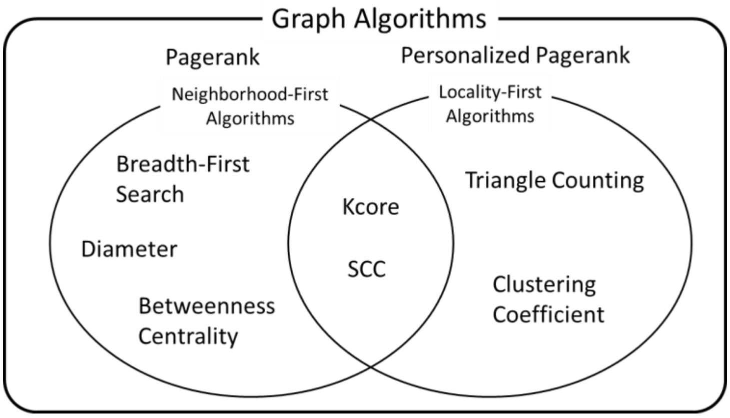


Figure 7 Classification of graph algorithms.

The Neighborhood-First algorithm is an algorithm in which the execution of the algorithm goes from vertex to neighbor vertex. These include BFS, Diameter, and Betweenness Centrality. Algorithms belonging to this class have several features. First, the neighborhood-first algorithm is performed through multiple iterations. Second, when the iteration is completed, the next iteration activates the vertex neighbors of the vertex that was performed in the previous iteration, and calls the vertex function. Third, in one iteration, a part of the whole vertex is activated. In order to improve the performance of this algorithm, assigning IDs to neighboring vertices firstly helps to improve I/O performance because I/O requests are made as close as possible.

The Locality-First algorithm is an algorithm in which the execution of the algorithm is performed and tends to generate I/O requests by locality equation (1). These include the Triangle Counting algorithm and the Clustering Coefficient algorithm. The algorithms belonging to this class have the characteristic that they have no iteration. That is, the whole algorithm operation is performed within one iteration. Also, since the vertex function is called at one vertex, the next operation generally tends to be that the vertex with high localization score is activated and the vertex function is called.

There are also algorithms that have both Neighborhood-First and Locality-First characteristics. There are Kcore and Strongly Connected Component algorithms.

Outside the diagram, there exists an algorithm that performs vertices in vertex ID order regardless of the topology of the graph. Algorithms such as PageRank and Personalized PageRank are usually linearly performed because the execution of vertices is performed in the ID order. Therefore, it is hardly affected by graph ordering. These algorithms are difficult to improve performance with graph ordering.

4. Performance Modeling

To find efficient graph ordering we need to know how performance will change when graph ordering is applied to the graph. So, graph ordering should be expressed numerically. In this chapter, we perform the modeling for graph ordering. In the previous chapter, each algorithm was classified according to the disk access pattern. Likewise, the modeling of the graph ordering is performed according to these classifications.

4.1. The Modeling for Neighborhood-First Algorithms

In this section, we perform I/O performance modeling for the Neighborhood-First allocation graph algorithm considering the analyzed graph algorithm operation. These algorithms include BFS, DIAM, and BC. These algorithms generate I/O requests at one vertex and then I/O requests at the next neighbor vertex. The I/O requests that occur after the operation of a specific vertex are as follows.

$$Next\ I/O\ requests\ = \sum I/O\ reqests\ of\ the\ neighbor\ vertex \quad (2)$$

Intuitively, when a large number of I/O requests are issued to the disk, the larger the distribution of the IDs of the requesting vertices, the similar to the random access. Conversely, the closer the distribution is, the more consecutive sequential accesses of I/O request vertex ID occur. In addition, the distribution of disk I/O requests as well as the number of pages requested will affect performance. Therefore, the I/O cost of one vertex can be calculated as shown in Equation (3).

$$I/O\ cost(v) = Deg(v) * \sigma^2(Neighbors(v)) \quad (3)$$

The I/O cost of the whole graph is defined by summing the I/O costs of each vertex. Equation (4) is the I/O efficiency of the graph.

$$I/O\ efficiency(G) = the\ inverse\ of\ \sum_{v \in V} I/O\ cost(v) \quad (4)$$

In order to verify whether the defined performance modeling reflects actual performance, we estimate modeling results and actual execution time of various graph orderings. Section 4.4 presents experimental results for modeling verification.

4.2. The Modeling for Locality-First Algorithms

In this chapter, we model the graph algorithms with Locality-First characteristics such as triangle counting and clustering coefficient. This modeling basically uses a method of calculating the locality that was used in the Gorder. Equation (5) below shows this.

$$S(u,v) = S_s(u,v) + S_n(u,v) \tag{5}$$

S_n = *whether two vertices are neighboring (0, 1, or 2)*

S_s = *whether two vertices are sibling (0 or 1)*

In equation (5), the locality between two vertices is calculated by taking two factors into consideration. First, if two vertices are adjacent to each other, there is a high probability that the next vertex will be performed after one vertex has been performed. It is represented by *Sn*. Second, if two vertices have a common neighbor vertex, then it is possible that two vertices, which are sibling vertices, are executed together after the common neighbor vertex is performed. This is expressed by the equation *Ss*.

The locality score for the entire graph is calculated as:

$$Locality\ Score(G) = \sum_{0<\emptyset(v)-\emptyset(u)\leq w} S(u,v) \tag{6}$$

In equation (6), we adopt a sliding window model with a window size w (w > 0) used in the Gorder. Assume two nodes, u and v, are assigned with IDs in the graph ordering, denoted as φ(u) and φ(v), and u appears before v (φ(v) > φ(u)).

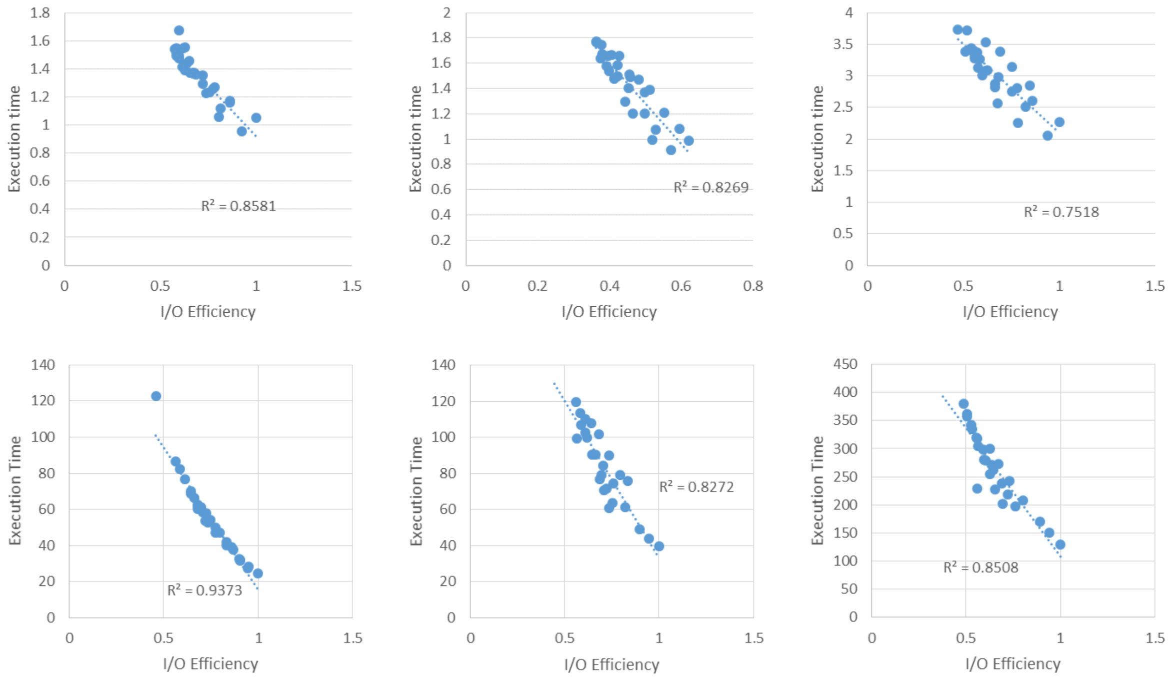
4.3. Combination

Equation (7) is a combination of the two formulas in Section 4.1 and Section 4.2. The graph algorithm may have both characteristics, or only one, depending on the case. Gorder is a graph ordering designed solely on the characteristics of Section 4.2. Therefore, in this paper, we propose a graph ordering for Neighborhood-First algorithm. This graph ordering has very low pre-processing costs and shows very high performance in Neighborhood-First algorithms and even reasonable performance in Locality-First algorithms.

$$Ordering\ efficiency(G) = \alpha I/O\ efficiency(G) + \beta Locality\ Score(G) \tag{7}$$

4.4. Verification

In this chapter, we do the verification about the modeling performed in the previous chapter. We estimate the costs estimated by the model and the actual execution times of the algorithms. A total of 25 graph orderings are applied. Linear regression was performed. The algorithms were performed on seven BFS, DIAM, BC, TC, CC, Kcore, and SCC. Alpha and Beta values are chosen as appropriate. Figure 8 shows the modeling and execution times when performing BFS (top) and TC (bottom). (The rest of the algorithm is currently doing experiments and will be added later.)



(a) Youtube (b) Flickr (c) Livejournal

Figure 8 Modeling verification.

In the linear regression analysis, R^2 values are highly correlated when they are close to 1 and closer to 0, indicating no correlation. Most R^2 value in Figure 8 is about 0.8, which indicates that the modeling results and performance are highly related.

5. Neighborhood Ordering

This section proposes a new graph ordering, Neighborhood Order (N.Order). Unlike Gorder designed for Locality-First algorithms, N.Order is designed to maximize the performance of Neighborhood-First algorithms. The modeling of Section 4.1 is determined by the degree of each vertex and the ID distribution of neighboring vertices. To maximize the modeling value, the following two factors should be minimized.

1) $Deg(v)$
2) $\sigma^2(neighbors(v))$

The first of these two elements is the topology of the graph and cannot be reduced, so the second element should be minimized. To reduce this, neighbor vertex IDs should be allocated as close as possible. The basic idea is to sequentially assign IDs to the neighbor vertices of the specific vertex. However, in the process of assigning IDs sequentially to neighboring vertices of a certain vertex, there is a possibility that the disk layout between neighboring vertices of some vertices is rather distant. Graph ordering, which minimizes the sum of the distances between neighboring vertices at all vertices, is very costly. So N.Order uses a way to approximate it. The start of N.Order is to select any target vertices and assign them sequentially to neighboring vertices.

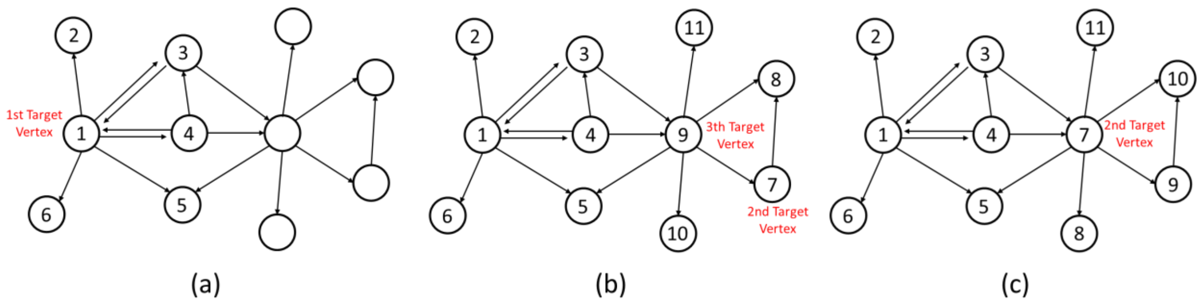


Figure 9 Procedure of graph ordering.

Jaccard similarity: The jaccard similarity is a method of measuring the similarity between two vertices **[10]**. This is represented by $|a \cap b|/|a \cup b|$. We use the jaccard similarity to reduce the variance between neighbor vertex ID. Figure 9 (a) shows the ID assigned to the neighbor vertex of the target vertex. In order to reduce the total variance, the next target vertex selection should be a vertex that has the most intersection with the already assigned vertex. At this time, we calculate jaccard similarity and select the target vertex with the highest value. Figure 9 (b) shows a case where the next target vertex is selected arbitrarily, and in the case (c), the vertex with the highest jaccard similarity value is selected as the target vertex. Calculating the I/O efficiency of the graph in both cases, (c) has a higher value.

However, calculating the jaccard similarity has a large overhead. To reduce this overhead, a simple and intuitive method is used. This intuition is that if there is a common neighbor vertex between two vertices, the jaccard similarity will be higher than that between randomly selected two vertices. Figure 10 shows the number of common vertices between any two vertices and the number of common vertices between two vertices that share the same vertex. It can be seen that the number of common vertices is much higher in the case of the same parent vertex sharing than in any case.

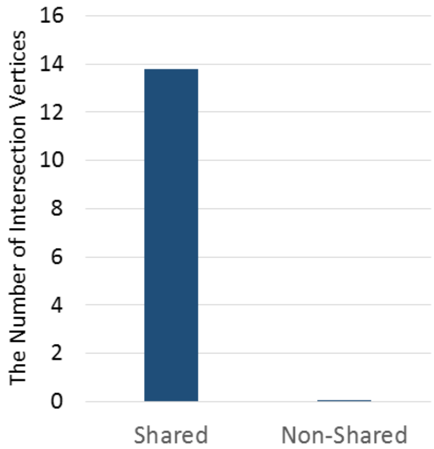


Figure 10 The number of common vertices.

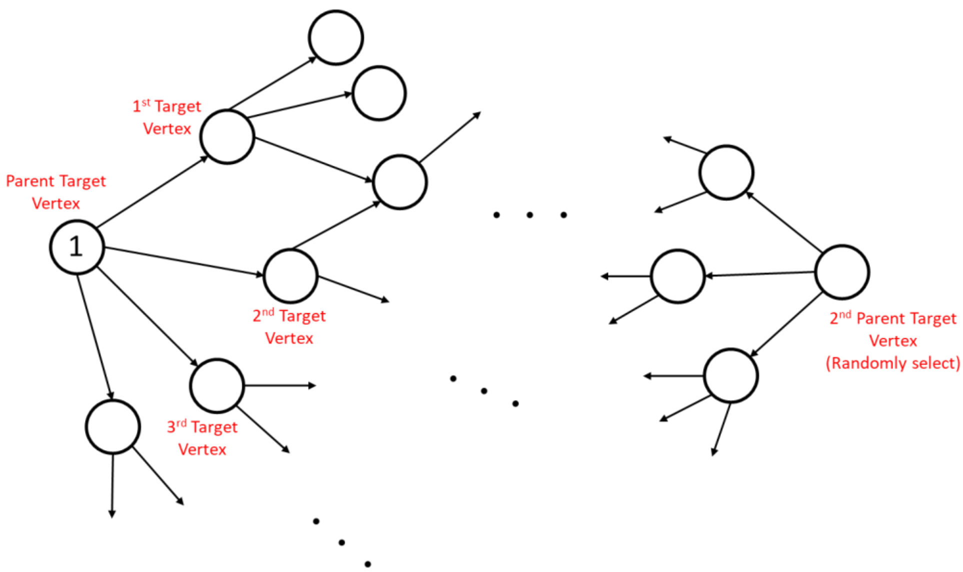


Figure 11 N.Order(Neighborhood Ordering)

N.Order uses two basic principles:

1) Sequentially allocate neighboring vertices of the target vertex.
2) Select one parent target vertex and sequentially select neighbor vertices as target vertices.

Figure 11 shows the allocation method of N.Order. The first parent target vertex is arbitrarily selected, and the neighbor vertices of this vertex are sequentially assigned the ID to the target vertex. When all the target vertices have been allocated, the next parent target vertex is randomly selected, and the ID allocation is repeated. This simple allocation scheme has very little pre-processing cost.

6. Evaluation

All the experiments are performed on a machine with Intel Xeon E5-2683 v4, running on Ubuntu 16.04. Intel 400GB SSD and HGST 500GB 7200 RPM HDD were used as external storage, and both disks are connected via SATA 6.0 Gb/s interface. For each experiment, only a single disk was used. The experimental environment is summarized in Table 1.

We have evaluated N.Order on FlashGraph, a semi-external graph engine optimized for SSDs. We choose FlashGraph because: 1) it is one of representative semi-external graph engines; 2) it is recently developed, thus most of known I/O optimizations are implemented; and 3) it is actively maintained, and core graph algorithms are already implemented in the system.

We experiment with four real-world networks that are publicly available shown in Table 2. Total seven core graph algorithms including Breath-First Search(BFS) **[11]**, Diameter(DIAM) **[12]**, Betweenness(BC) **[13]**, PageRank(PR) **[14]**, Personalized PageRank(PPR) **[15]**, Triangle Counting(TC) **[16]**, Clustering Coefficient(CC) **[17]** are evaluated.

Table 1 Experimental environment

Software	Ubuntu 16.04 LTS (kernel 4.4.0-38)
	FlashGraph 0.3.2
Hardware	Intel Xeon E5-2683 v4
	Intel 400GB SSD
	HGST 500GB 7200RPM HDD

Table 2 Datasets for the evaluation [18]

Graph	|*V*|	|*E*|	***Size(GB)***
Youtube	3,223,589	9,375,374	0.26
Flickr	2,302,925	33,140,017	0.42
Livejournal	4,847,571	68,475,391	1.02
Wikipedia	18,268,992	172,183,984	2.6

6.1. Execution Time

Firstly, to verify the performance of N.Order, we measured performance along with various ordering policies, including the latest graph ordering, Gorder. The ordering policy used in the experiment is as follows.

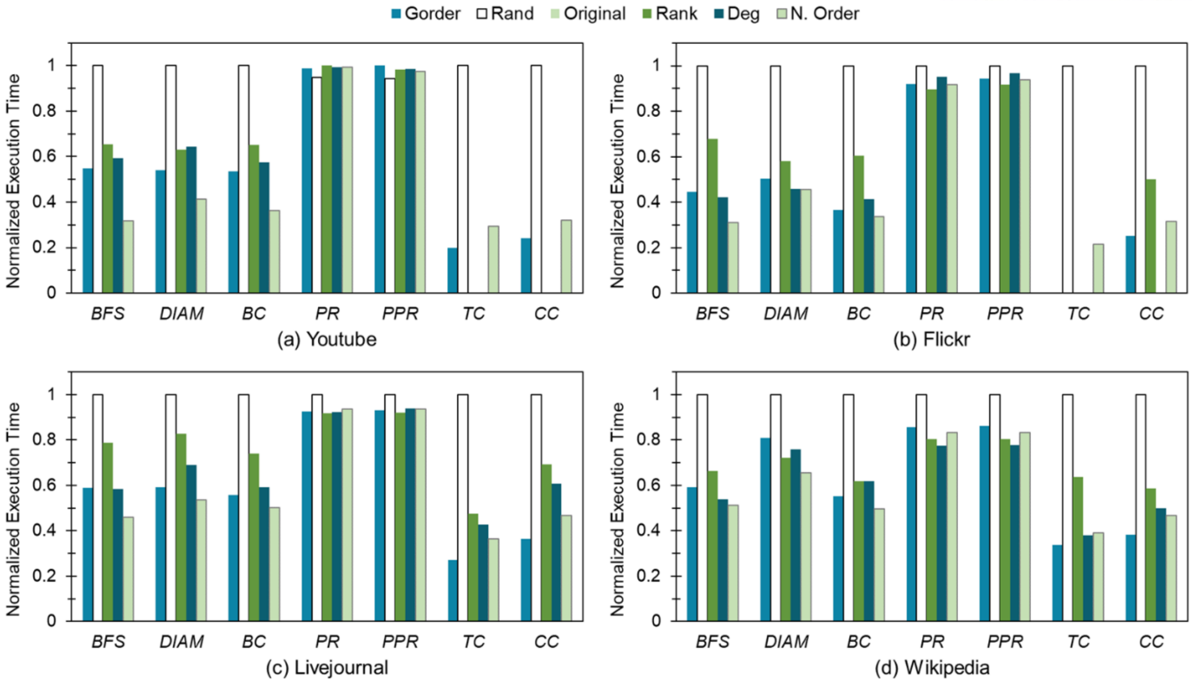


Figure 12 Normalized execution times of graph algorithms on SSD with various graph orderings.

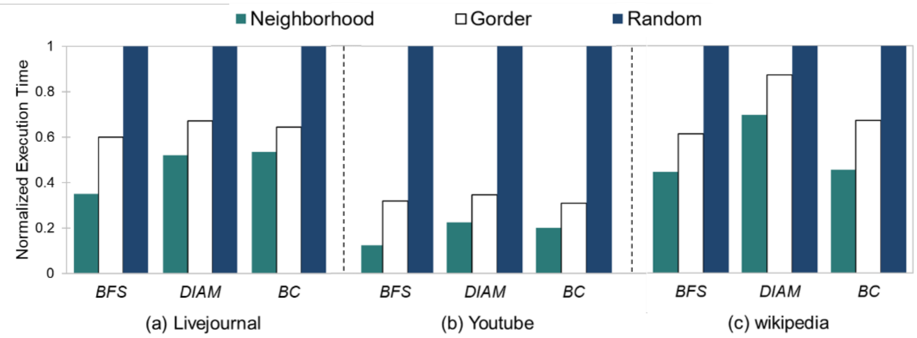


Figure 13 Normalized execution times of Neighbor-First algorithms on HDD with three graph orderings

- Gorder: The latest graph ordering
- Random: Assign ID randomly
- Rank Sort: Assigned in descending order of PageRank value
- Deg. Sort: Assign IDs in descending order of degree
- N.Order: The proposed ordering policy

Figure 12 shows the execution time results of the graph algorithms for each ordering in the SSD. Unmarked bars are algorithms that cannot be executed with a 10% size of page cache. In the case of the

Neighborhood-First algorithm, N.Order proposed in this paper showed the highest performance, and the performance was improved about twice as much as Random. On the other hand, in the case of Locality-First Algorithm (TC, CC), the Gorder which reflects the locality has the best performance. In the case of PR and PPR, there is no performance improvement due to the limitation of section 3.1.

The impact of graph ordering is more significant when HDDs are used as the external storage. Figure 13 compares three graph orderings – neighborhood ordering, Gorder, and random ordering – for BFS-like algorithms. The size of the page cache is set to be 10% of the size of the input graphs. We can immediately notice that the performance difference is much larger. Execution with neighborhood ordering is up to an order of magnitude faster than random ordering and two times faster than Gorder. As HDD is slower than SSD and their performance for random access is much worse, the impact of graph ordering becomes more pronounced with HDD.

In summary, N.Order improves performance of graph algorithms running on disk-based graph engines by a factor of two on SSD and by a factor of ten on HDD.

6.2. I/O Request Size

Figure 14 shows the number of I/O requests per ordering when performing the BFS algorithm. When disk I/O occurs in adjacent IDs, I/O requests are merged, and I/O is efficiently handled. Therefore, in the case of N.Order, the size of I/O request is smallest compared with other ordering policies due to the merge effect.

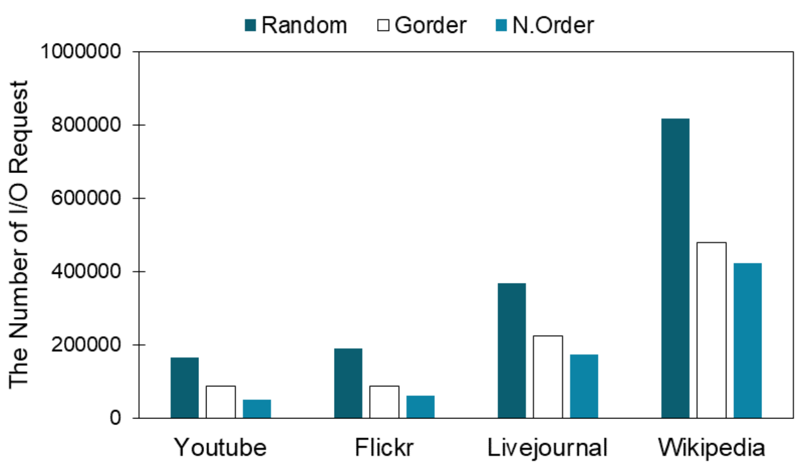


Figure 14 I/O request size of BFS algorithm with Random, Gorder and N.Order.

6.3. Pre-processing Overhead

Table 3 shows the ordering costs of Gorder and N.Order. As a result, N.Order shows that the preprocessing cost is at least 6.2 times less than the Gorder.

Table 3 Computation times in seconds
for Gorder and Neighborhood Ordering (denoted as N.Order)

	Youtube	Flickr	Livejournal	Wikipedia
Gorder	12.5	39.6	45.6	169.3
N.Order	2.0	2.7	7.2	16.9

7. Conclusion

The evolution of the web and social networks has naturally necessitated large-scale graph processing. When using a disk-based graph engine to handle large graphs, effective I/O workload control has a significant impact on performance. In this paper, we analyze the I/O workload of each graph algorithm in disk-based graph engine and model the relationship between graph ordering and performance. Based on this, we propose a new graph ordering technique called N.Order. The proposed N.Order reduces the ordering overhead by at least 6.2 times compared with the recently proposed Gorder, and it shows a performance up to 10 times higher than random ordering for Neighbor-First graph algorithms. Future research is the development of graph-specific user-level file system that utilizes the graph ordering proposed in this paper.

8. Related Works

Disk-Based Graph Engines: GraphChi is the first disk-based graph processing system that made possible large-scale graph analysis on a single machine. GraphChi is primarily designed for HDDs and eliminates most random disk accesses with its Parallel Sliding Windows technique.

TurboGraph is a disk-based graph engine optimized for SSDs. In TurboGraph, adjacency lists of requested vertices are randomly accessed in each iteration. Its pin-and-slide execution model overlaps random I/O with CPU computation and fully utilizes the I/O parallelism inherent within SSDs. Because TurboGraph makes use of page cache for its random I/O, our optimizations can be simply applied to improve its performance.

While the vertex-centric computation model is widely adopted in large-scale graph processing, recent studies on disk-based graph systems propose an alternative edge-centric computation that streams edges into memory to exploit the locality of edge access [19, 20]. The edge-centric model shows good performance for algorithms that access the entire graph repeatedly. However, the model is not as efficient for algorithms that iteratively access different subsets of a graph as is done with BFS-like and subgraph mining algorithms.

In semi-external graph engines, vertex attributes are stored in main memory for fast updates [6, 21, 22, 23]. Pearce et al. proposed asynchronous optimization techniques for graph traversal algorithms for semi-external graph processing [22]. FlashGraph implements several I/O optimizations for SSDs and SSD arrays such as reducing the amount of I/O with compact graph representation, merging I/O requests for higher throughput, and overlapping I/O and computation [6]. We build on top of these optimizations and propose optimizations that exploit the structural properties of the input graph.

Several other I/O optimization methods for disk-based graph processing have recently been proposed. Vora et al. employs a dynamic partitioning scheme that prevents loading unnecessary edges on disk [7]. GridGraph supports 2D edge partitioning to reduce I/O access [20]. In Graphene, a bitmap based asynchronous I/O optimization is applied to efficiently merge small I/O requests [24]. Our proposed optimization techniques are applicable on top of these I/O optimizations.

Parallel and Distributed Graph Processing. Google's Pregel pioneered large-scale graph processing with the vertex-centric computation model [1]. GraphLab is a distributed machine learning framework that employs a variant of vertex-centric model that supports asynchronous computation [3]. PowerGraph, a successor of GraphLab, adopts an efficient graph partitioning scheme that considers the power-law degree distributions of real-world graphs [2]. GraphX transforms graph processing operations in the vertex-centric model to dataflow operations such as join, map, and reduce [25].

To ease the programming difficulty of large-scale graph analysis, SociaLite supports declarative query language based on Datalog [26]. Wang et al. also demonstrate the performance and scalability of Datalog-based graph processing [27]. It should be simple to support declarative graph processing with the optimizations we proposed in this paper.

Galois is a parallel graph processing system based on an implicitly parallel vertex iterator [28]. Green-Marl is a domain-specific language for writing parallel graph algorithms for shared-memory [29]. Ligra is a light-weight framework that provides low-level primitives to implement graph processing systems [30].

Wei et al. studied optimizing the performance of graph algorithms for main memory graph processing [8]. The authors develop a graph ordering called Gorder that optimizes the locality of updating vertex attributes. While Gorder is designed for main memory graph systems, neighborhood ordering is designed to optimize the performance of disk-based graph engines.

Appendix

1. Parallel graph processing: Is random ordering always the worst?

In this paper above, the performance of random ordering always shows the worst performance. If so, is random ordering the worst graph ordering? Not always. The random ordering is the worst in terms of the cache locality of the graph. However, in multithreading graph processing, there is an advantage in terms of load balancing.

Multithreaded parallel computation for high-speed graph processing is the most commonly used. Multithreaded-based graph processing is typically implemented on various graph engine range-based partitioning to minimize partitioning overhead. Since the Gorder was designed as a single-threaded architecture, load balancing was not considered for multithreaded execution. Figure 15 shows the number of edges allocated to 8 multithreads when the partition size is 131072(2^{17}) in the livejournal graph and the execution time per thread when executing the PageRank algorithm. The experiment was performed assuming that there is no load-balancing mechanism in the graph engine.

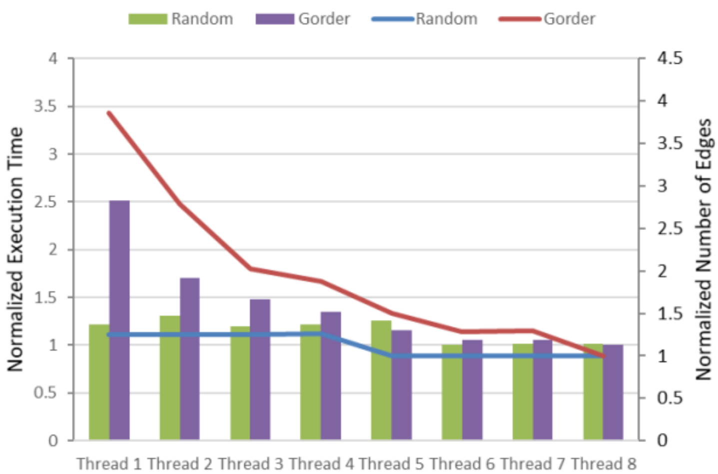


Figure 15 Number of allocated edges per thread (line) and per-thread execution time (bars) when performing PageRank.

In the case of Gorder, the number of edges per thread and the execution time are up to 3.5 times and 2.5 times, respectively. On the other hand, because random ordering assigns each vertex at random, the execution time and the number of edges per thread are determined to be stochastically similar. This load imbalance in the Gorder leads to performance degradation. Therefore, if a Gorder allocates an ID considering load balancing, it will show a bigger performance improvement.

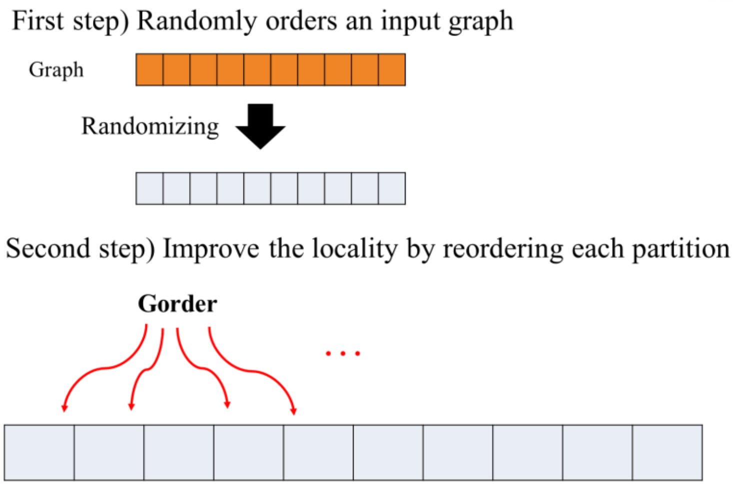


Figure 16 B.Gorder (Balanced Gorder)

We propose B.Gorder (Balanced Gorder) which is a simple ordering method by combining random ordering and Gorder instead of redesign of Gorder. Figure 16 shows how B.Gorder works. B.Gorder consists of two steps. First, load balancing is performed for each partition through random ordering in the graph. Second, by performing a Gorder on each partition, it improves the per-thread locality.

In order to check whether B.Gorder has load balancing effect, we measured execution time per thread when executing PageRank in multithread environment. Figure 17 shows the execution time of each thread when Gorder and B.Gorder are applied respectively. In Gorder, the execution time deviations per thread were at least 2.5 times up to 13 times.

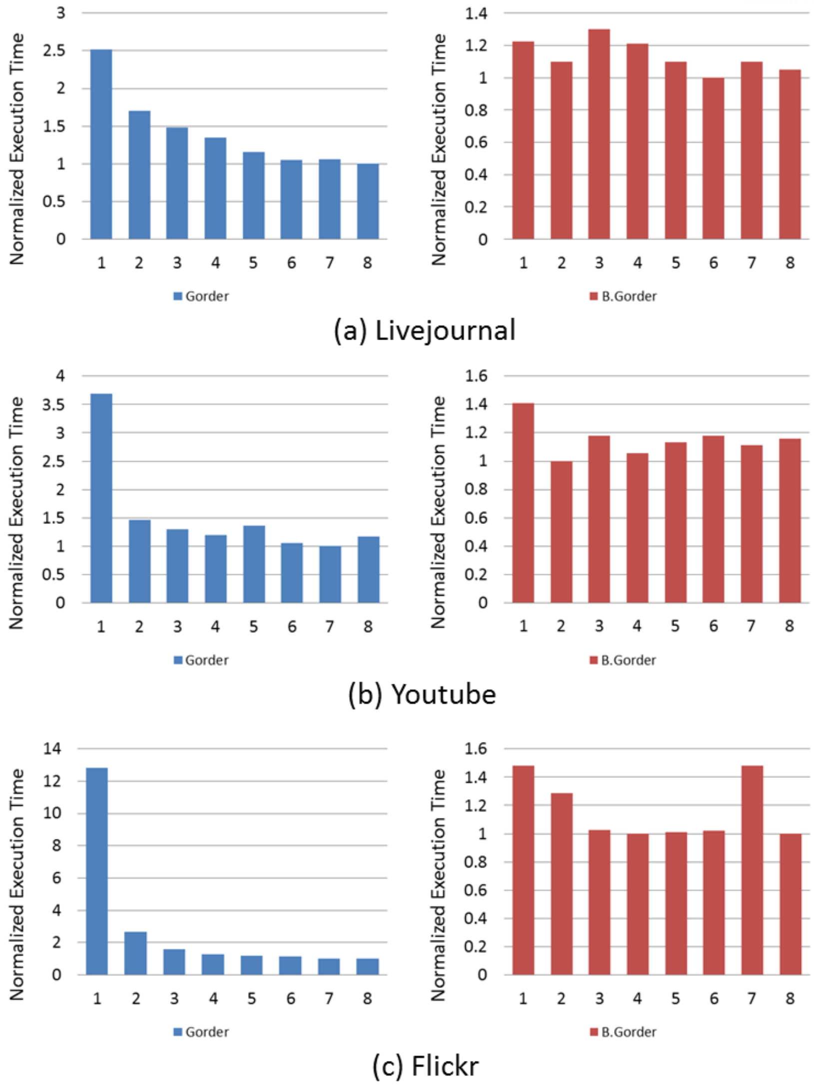


Figure 17 Execution time per each thread in the PageRank algorithm.

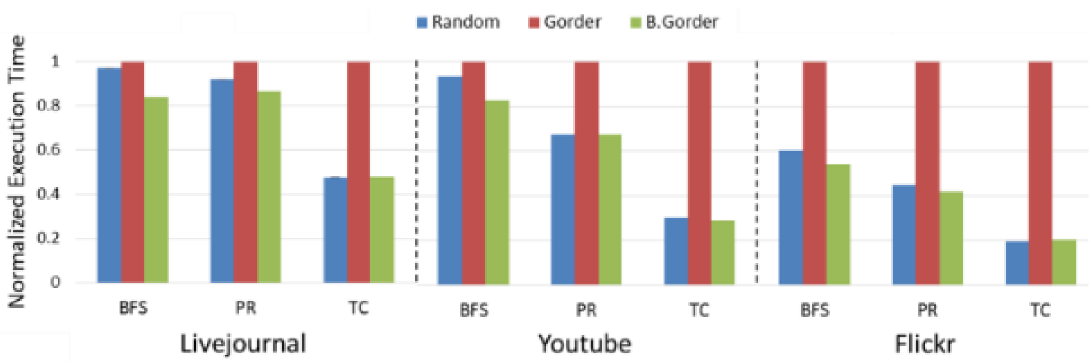


Figure 18 Execution time for three ordering in multithreaded environments.

We also measured performance for three key algorithms to measure the impact of load balancing on overall performance. Figure 18 shows the execution time of the three core algorithms for each ordering in a multithreaded environment. Gorder has very low performance due to load imbalance in multithreaded environment. Compared with Figure 17, it can be seen that the worse the performance is, the more severe the load imbalance is. B.Gorder shows a performance improvement of at least 15% to 5 times higher than the Gorder.

The random ordering scheme can sometimes be a good choice if the load balancing is not implemented in the graph processing engine and the imbalance is severe. In conclusion, the idea that "random ordering is always the worst ordering" is partially inadequate.

2. Toward zero pre-processing costs

Graph ordering is a pre-processing process independent of algorithm execution. As the size of the graph increases, the cost of pre-processing also rises proportionally. The time complexity of the Gorder is approximately O (n^2). Because the actual graph size is beyond Billion, the Gorder with high time complexity is hard to apply in practice. Therefore, this chapter presents one idea called page-level ordering to further reduce pre-processing costs. The proposed idea can reduce the pre-processing time to a very large extent because it reduces the size of the graph to be applied to the graph ordering to several hundredths of a degree.

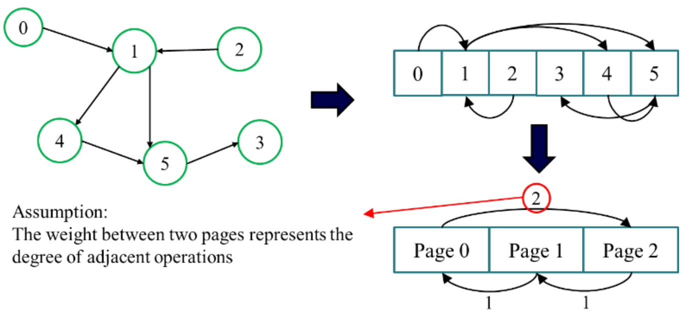


Figure 19 Basic principles of page-level ordering

Figure 19 shows the basic principles of the proposed idea. In Figure 19, the graph on the left is listed in the order in which they are stored on the disk, which is expressed as a linear representation in the upper right corner. Since I/O is performed page by page, the expression on the upper right is re-expressed in page-level graph below. The weight shown above the edge between pages is the number of edges between pages. The larger the weight, the more likely the two pages will be used together.

Therefore, it is better to place pages with large weights adjacent to each other. Also, if one edge is 4 bytes, one thousand edges are stored in one page. Thus, the size of the newly created page-level graph is one thousandth of the original graph. Because the graph size to be ordered is very small, the cost of preprocessing is expected to be greatly reduced.

REFERENCES

[1] G. Malewicz, M. H. Austern, A.J.C Bik, J. C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, "Pregel: A System for Large-Scale Graph Processing," Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, pp. 135-146, 2010

[2] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs," Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation, pp. 17-30, 2012

[3] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, J. M. Hellerstein, "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud," Proceedings of the 38th International Conference on Very Large Databases, pp. 716-727, 2012

[4] A. Kyrola, G. Blelloch, C. Guestrin, "GraphChi: Large-Scale Graph Computation on Just a PC," Proceedings of the 10th USENIX Simposium on Operating Systems Design and Implementation, pp. 31-46, 2012

[5] W. S. Han, S. Lee, K. Park, J. H. Lee, M. S. Kim, J. Kim, H. Yu, "TurboGraph: A Fast-Parallel Graph Engine Handling Billion-scale Graphs in a Single PC," Proceedings of the 19th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, pp. 77-85, 2013

[6] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, "FlashGraph: Processing Billion-Node Graph on an Array of Commodity SSDs," Proceedings of the 13th USENIX Conference on File and Storage Technologies, pp. 45-58, 2015

[7] K. Vora, G. Xu, R. Gupta, "Load the Edge You Need: A Generic I/O Optimization for Disk-based Graph Processing," Proceedings of the 2016 USENIX Annual Technical Conference, pp. 507-522, 2016

[8] H. Wei, J. X. Yu, C. Lu, X. Lin, "Speedup Graph Processing by Graph Ordering," Proceedings of the 2016 International Conference on Management of Data, pp. 1813-1828, 2016

[9] R. R. McCune, T. Weninger, G. Madey, "Think Live a Vertex: A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing," ACM Computing Surveys (CSUR), Vol. 48, No. 25, pp. 1-39, 2015

[10] Jaccard similarity(index), https://en.wikipedia.org/wiki/Jaccard_index/

[11] Breadth-First Search, https://en.wikipedia.org/wiki/Breadth-first_search/

[12] Diameter, http://mathworld.wolfram.com/GraphDiameter.html/

[13] Betweenness Centrality, https://en.wikipedia.org/wiki/Betweenness_centrality/

[14] Page Rank, https://en.wikipedia.org/wiki/PageRank/

[15] G. Jeh, J. Widom, "Scaling personalized web search," In WWW Conference, pp. 271-279, 2003/

[16] Triangle Counting, http://theory.stanford.edu/~tim/s14/l/l1.pdf/

[17] Clustering Coefficient, https://en.wikipedia.org/wiki/Clustering_coefficient/

[18] The koblenz network collection, http://konect.uni-koblenz.de/

[19] A. Roy, I. Mihailovic, W. Zwaenepoel, "X-Stream: Edge-Centric Graph Processing using Streaming Partitions," Proceedings of the 24th ACM Symposium on Operating Systems Principles, pp. 472-488, 2013

[20] X. Zhu, W. Han, W. Chen, "GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning," Proceedings of the 2015 USENIX Annual Technical Conference, pp. 375-386, 2015

[21] P. Kumar, H. H. Huang, "G-Store: High-Performance Graph Store for Trillion-Edge Processing," Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 830-841, 2016

[22] R. Pearce, M. Gokhale, N. M. Amato, "Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory," Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1-11, 2010

[23] Z. Shao, J. He, H. Lv, H. Jin, "Fog: A Fast Out-of-Core Graph Processing Framework," International Journal of Parallel Programming, pp. 1-14, 2016

[24] H. Liu, H. H. Huang, "Graphene: Fine-Grained I/O Management for Graph Computing," Proceedings of the 15th USENIX Conference on File and Storage Technologies, pp.285-300, 2017

[25] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, "GraphX: Graph Processing in a Distributed Dataflow Framework," Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, pp. 599-613, 2014

[26] J. Seo, J. Park, J. Shin, M. S. Lam, "Distributed SociaLite: A Datalog-Based Language for Large-Scale Graph Analysis," Proceedings of the 39th International Conference on Very Large Databases, pp. 1906-1917, 2013

[27] J. Wang, M. Balazinska, D. Halperin, "Asynchronous and Fault-Tolerant Recursive Datalog Evaluation in Shared-Nothing Engines," Proceedings of the 41th International Conference on Very Large Databases, pp. 1542-1553, 2015

[28] D. Nguyen, A. Lenharth, K. Pingali, "A Lightweight Infrastructure for Graph Analytics," Proceedings of the 24th ACM Symposium on Operating Systems Principles, pp. 456-471, 2013

[29] S. Hong, H. Chafi, E. Sedlar, K. Olukotun, "Green-Marl: A DSL for Easy and Efficient Grapn Analysis," Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 349-362, 2012

[30] J. Shun, G. E. Blelloch, "Ligra: A Lightweight Graph Processing Framework for Shared Memory," Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming, pp.135-146, 2013